1. 上课须知

课程主题：HBase 分布式 NoSQL 数据库--第一次课（架构设计完全剖析）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次内容总结

ZooKeeper四次课：

```
1、分布式理论相关
    集中式单体服务架构 + 分布式服务部署架构
    分布式系统常见问题探讨 + 分布式一致性理解
    分布式事务2PC + 3PC
    分布式一致性算法：Paxos Raft ZAB
    鸽巢原理 + QuorumNWR理论
    CAP 和 BASE 理论
2、ZooKeeper架构设计
    zookeeper的znode系统详解
    zookeeper的watch机制详解
    zookeeper的应用场景和设计目的
    zookeeper的应用案例
        利用zookeeper实现分布式锁
        利用zookeeper实现选举
```

```
        利用zookeeper实现配置管理
3、zookeeper源码剖析
    四大支撑：
        序列化机制
        持久化机制
        网络通信机制
        watcher监听机制
    两个核心源码流程：
        启动
            如何从磁盘恢复znode系统
            如果选举leader-->follower和leader的状态同步
        读写
            zookeeper.create()
            zookeeper.getData()
    QuorumPeerMain启动解析
    ZKDataBase恢复解析
    FastLeaderElection选举解析
        startLeaderElection() 为选举做准备
4、zookeeper源码分析
    FastLeaderElection选举解析
        lookForLeader() 选举执行
    状态同步
        Leader.lead()                领导状态
        Follower.followLeader()      状态同步
```

3. 上次作业

作业：实现类 ZooKeeper 数据模型系统

```
1、基础的 树形结构 和 CURD 功能
2、具备冷启动数据恢复/状态恢复的能力
```

4. 本次内容预告

本次的课程内容是：HBase 分布式 NoSQL 数据库

需求背景：

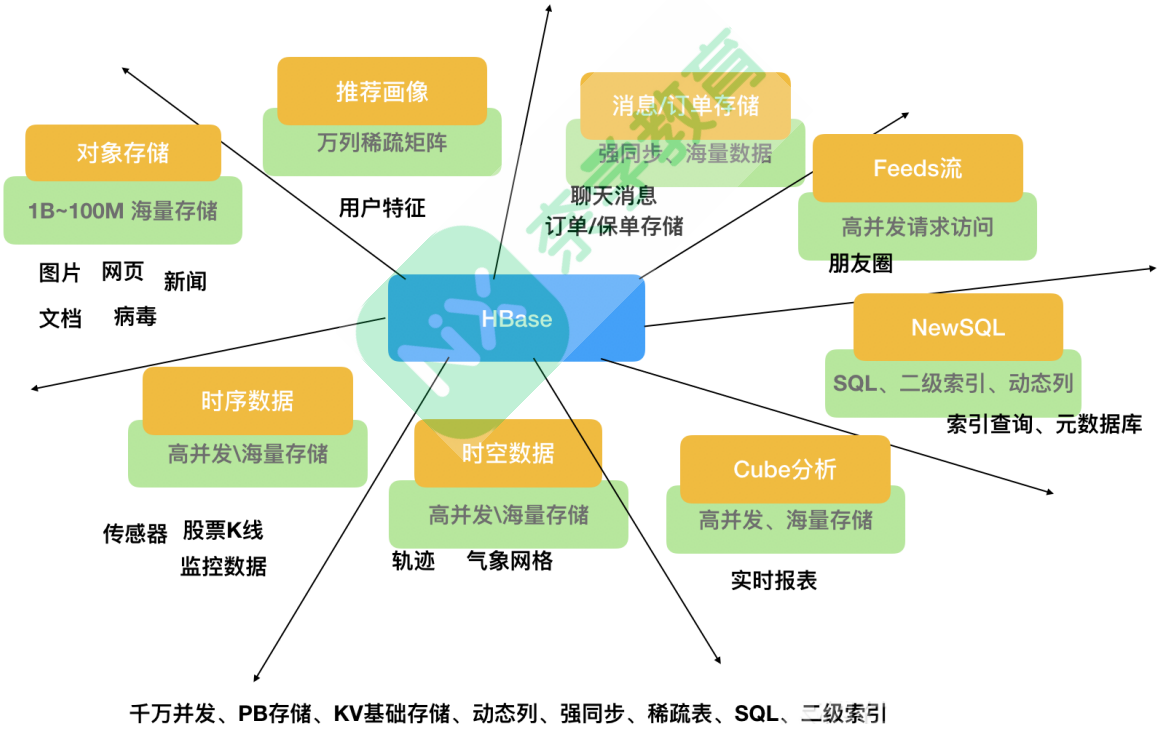
1、两个基础的组件：
    HDFS， MySQL ==> 基于HDFS实现一个分布式数据库 ==> 解决从海量数据中做低延时的随机读写！
2、需求背景
    谷歌 nutch 爬虫 两大核心需求：海量网页的解析计算 + 解析计算结果的存储问题
    提供解决方案：
    1、解决海量网页存储的问题：HDFS
    2、解决海量网页的解析计算：MapReduce(pageRank + 倒排索引)
    3、解决存储这些解析结果的问题：BigTable（解决海量key-value数据（用户搜索的关键词--URL）的存储）
        用户输入搜索关键词，系统从BigTable中，根据搜索关键词作为key去寻找 URL，最后根据每个URL的score进行排名展示
    同时这三个分布式技术都提到了一个共同的组件：chubby 开源实现就是 zookeeper

核心问题：**如果是你，你需要一个能在海量数据集中，做低延时的随机读写操作，你该怎样设计这个系统呢?**

简化一下该问题：假设我有100PB 的数据，你需要在0.1s以内找出一条数据。

HBase解决的核心问题：**海量数据的低延迟的随机读写**

HBase的应用场景：



4.1. 本次内容概述

01、如何设计一个分布式数据库头脑风暴
02、高效的增删改查数据结构和算法解析
03、HBase的架构设计
04、HBase的核心概念
05、HBase的表逻辑模型
06、HBase的Java API设计
07、HBase的核心工作机制读写流程解析
08、HBase的Flush,Split和Compact（源码剖析的时候，详细讲解）

4.2. 如何设计一个分布式数据库头脑风暴

一个小问题：我心里面想了一个数字，范围在1-100之间，你使用什么方式能最快确定这个数是多少？

思路引爆点：

```
1、怎样快速判断一个元素在不在一个数据集中？（布隆过滤器）
2、是否能找到一种合适的数据结构和搜索算法能快速从一个数据集中找出一个元素？（二分查找）
3、如何设计分布式系统？网络编程模型（NIO RPC Netty）
4、是否可以提前过滤不参与查询的数据，提高查询效率？（列裁剪，列式存储，谓词下推）
```

需求：确保在海量数据中，低延时的随机读写

```
1、内存 + 磁盘
    必然不能进行数据读写时的顺序追加，先写入数据到内存，内存空间装到一定程度的时候，先进行排序，然后溢写到磁盘文件。多次溢写就会形成多个磁盘文件。针对这些磁盘文件做归并(文件合并 + 保证顺序)
2、内存数据良好的数据结构
    这一块内存数据，在溢写之前，是没有在磁盘中存在的！
    1、如果遇到了掉电问题：内存丢失    WAL（每次写入一条数据到内存的先记录操作日志，数据再写到内存）
    2、如果遇到了查询需求：内存中的数据，也有插入需求，也有查询需求，你必须得设计一种数据结构去提高查询和插入效率：这个内存结构在HBase也是用的 跳表（ConcurrentSkipListMap）O(LgN)
3、磁盘数据 + 布隆索引
    最大的作用，就可以帮助用户快速判断，要搜寻的数据，在不在需要查找的数据分段中
4、排序 ==> 二分查询 ==> 范围分区 + 索引 ===> 跳表    HBase 的核心设计思路
5、跳表Topo结构
6、读缓存 + 写缓存
    为了提高数据写入效率：数据先写入到内存
    为了提高查询效率：把热点查询数据，放到读缓存中。
7、WAL机制 -- LSTM-Tree
```

需求：确保在分布式架构中，保证数据的安全

```
1、内存 + 磁盘  ----> 提高效率，保证安全
2、WAL机制  -----> 保证安全
```

HBase核心介绍：HBase 是 Google 的 BigTable 的开源实现，底层存储引擎是基于 LSM-Tree 数据结构设计的。写入数据时会先写 WAL 日志，再将数据写到写缓存 MemStore 中，等写缓存达到一定规模后或满足其他触发条件才会flush刷写到磁盘，这样就将磁盘随机写变成了顺序写，提高了写性能。每一次刷写磁盘都会生成新的HFile文件。

核心思路和过程：

1、解决海量数据的低延时的随机读写需求。必然需要给这些数据进行排序，假设这些数据就都是存储在一张表中。
2、既然数据有序，就可以根据二分查询的思路，去构建一个多层索引的跳表结构，最大的好处，就是可以在固定的时间里面，快速把待搜寻的数据范围降低到原来的1/n，n就是这张表的数据分段个数。这张表排序了，分成了多段，范围分区
　　隐藏一个问题：假设一开始你就把这张表分成了100段。在不停的插入数据的过程中，每个分段会越来越大
　　解决方案：在每个分段数据越来越大的时候，当然要保证这个分段的数据有序，然后把这个分段的数据，一分为二
3、分段中的数据，有三种组织形式：写缓存，读缓存，磁盘文件
4、先到读缓存中，查询，如果命中，则直接返回
5、如果没有命中，再去写内存中查找，写内存被设计了一种特殊的数据结构能够帮助提高插入和查询的效率，所以如果能命中，这个效率也很高
6、如果写内存也没有命中，最后就只能到磁盘文件去搜寻
7、去设计一个布隆索引：快速判断一个元素是否在这个文件中。
8、如果磁盘文件有多个，我使用布隆索引，挨个儿询问，如果要搜寻的数据在某一个文件中（布隆返回true），这个文件一般来说，都不会被设计很大的。这个文件，就是跟HDFS的默认的数据块的大小是一样的。128M
9、这个128M 的文件也会被精心设计，来提高查询效率，这个文件名称叫做HFile，自带索引结构。到时候再讲

核心的查询步骤：

1、需要从跳表体系中，定位数据分段，　这种体系决定了，不管数据量多大，都能在固定的时间长度内返回结果
2、读缓存　这个缓存大小不是特别大。
3、写缓存　一种独特的数据结构：　跳表
4、磁盘文件
　　通过布隆索引来判断，到底在那个文件
　　扫描文件：HFile 自带索引，其实很快速

4.3. 高效的增删改查数据结构和算法解析

通过刚才的思路引导，至少知道有很多的方式在能保证数据安全的情况，也能保证海量数据集中的低延时的随机读写操作。如果是查询分析呢?

Hive
HDFS + MapReduce
HBase + MapReduce
以上三种方案的效率都很低
解决方案：
1、优秀的基于内存的DAG分布式计算引擎
2、一些优秀的专门被设计用来做查询分析的存储系统

有一些需求：TP　事务操作：单条记录的增删改 HBase 设计目的用来解决在海量数据中，做低延时的根据key找value

有一些需求：AP　查询操作：select.... group by ....　(kylin doris clickhouse)

HDFS Kakfa ZooKeeper HBase（非常庞大复杂的map）　存储

Kylin Doris ClickHouse Kudu Druid + Flink 构建实时数仓!

MySQL 被设计出来用来干什么的？ insert + select 针对单条记录的正删改查操作 select ... groupby ....

4.4. HBase的架构设计

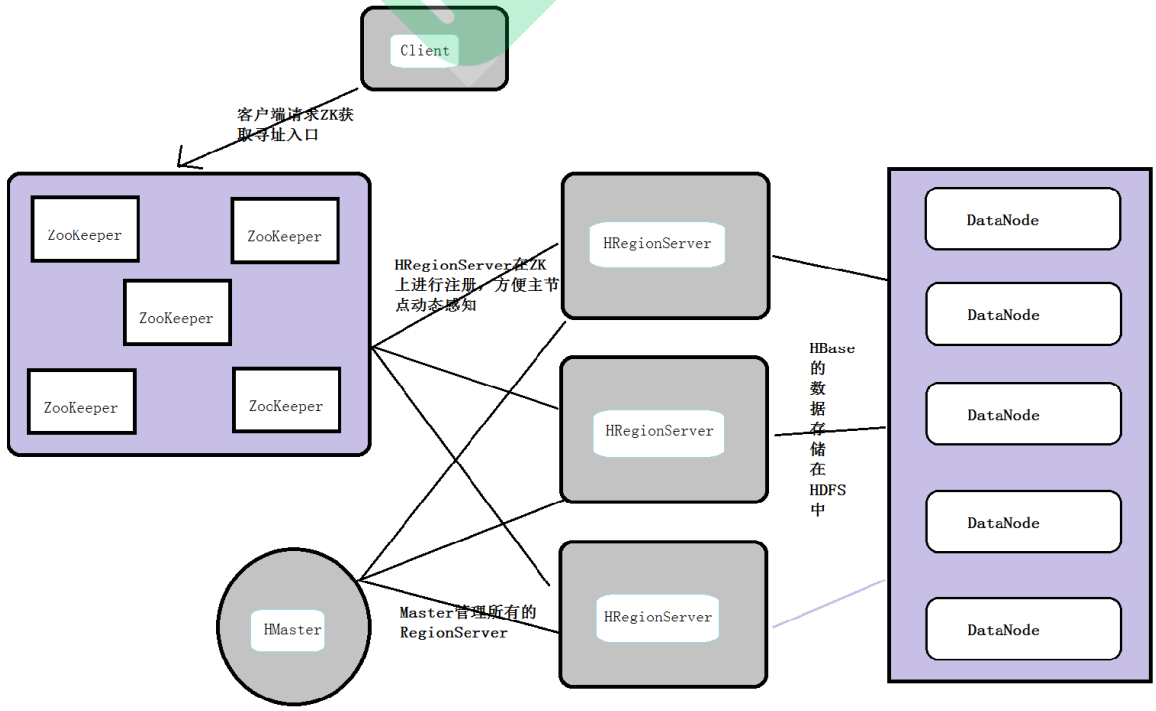
HBase的架构图：

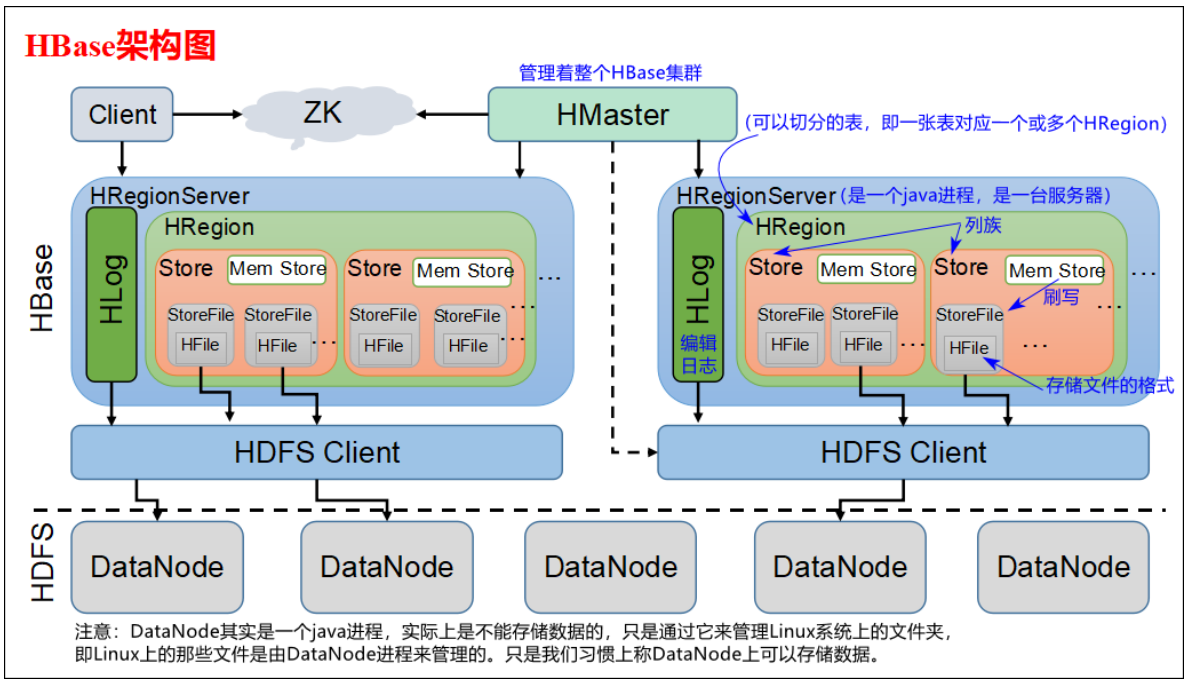
```
client        客户端，可以缓存regoin的位置信息
master            集群的主节点
regionserver          集群的工作节点
zookeeper         hbase的服务协调组件 也是HBase表的寻址入口
hdfs          hbase生成的数据文件是通过HDFS来进行委托存储的
```

```
文件    分布式文件       HDFS    file        block
表      分布式表         hbase    table     region      数据会进行排序的，分区之后，就变成了范围分区
集合    分布式集合       spark    rdd         partition
```

```
hbase-0.96x 以前，这个跳表结构是一个三层结构：
-root-  无论如何不会被拆分， 这个表始终只有一个region，这个唯一的region 的位置信息，被存储在 zk 当中
.meta.
usertable
```

```
hbase-0.96以后：把 -root- 这个表去掉了。
    把每个region由256M 提升到了 10G (40倍)
干脆把 -root- 表去掉了， 然后 .meta. 表的几个regioin 的位置信息，就被存储在 zk 中
```



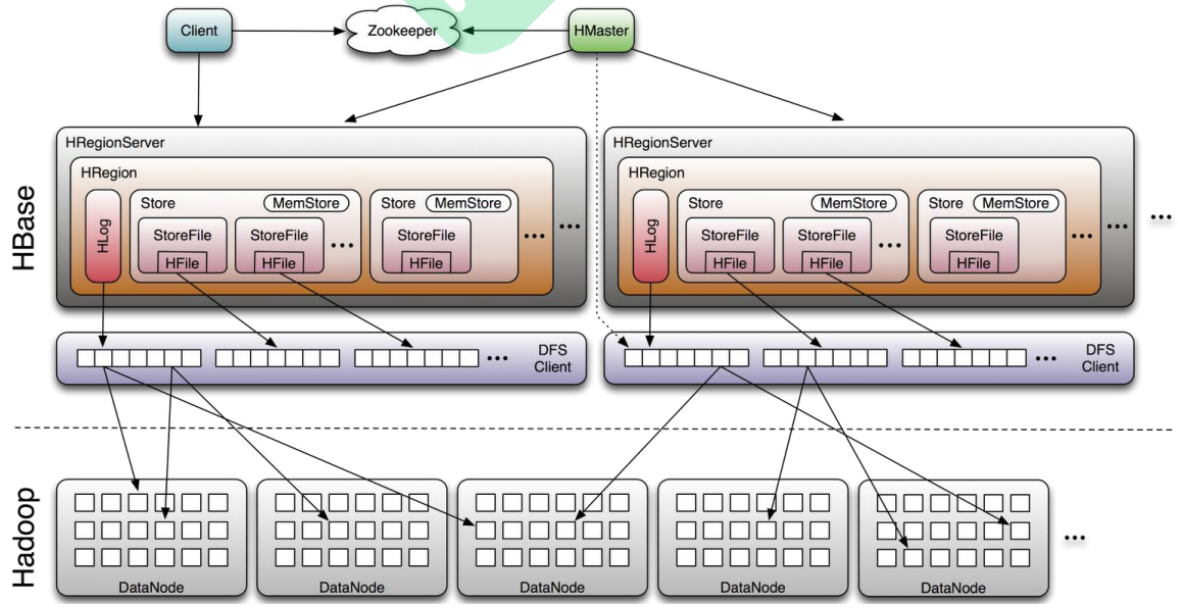


HBase的整体架构：HBase读写操作过程中，HMaster是不参与的！ 所有的元数据的操作，都要经过 master 来实现，当然大量的元数据信息，都被保存在了 ZK 中。 所以其实就是通过 master 来操作 zk 改变元数据

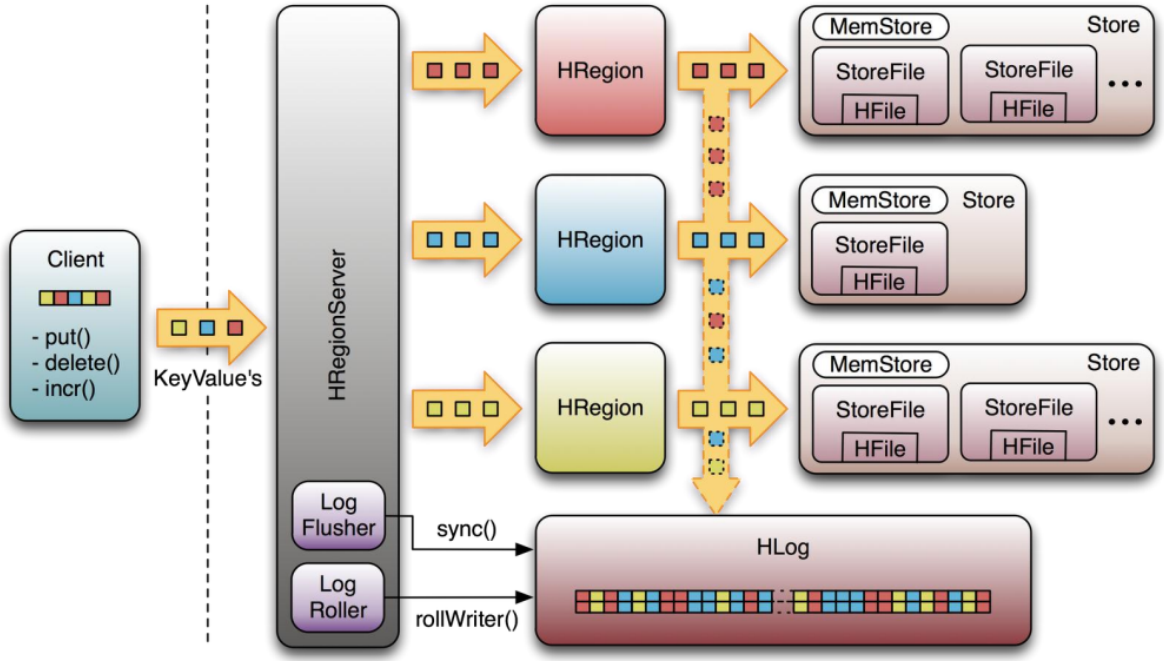
元数据包含两个方面：这两种元数据的操作，都是通过 master 来操作

```
1、表的 namespace  名称空间  table
    这种数据保存在 zk 中的

2、一张表到底包含了那些regionserver上面管理的那些region   这个映射信息
    这种数据保存在 meta 表中的
```



HBase中的HRegionServer的架构设计：




```
一个hbase集群有多个hregionserver
一个hregionserver管理了多个regoin
一个region内部有可能有个store
每个store里面有：
    memstore
    blockcache
    storeFile  HFile
一张表在横向方向上，按照行健 rowkey 进行范围分区，拆分成多个region
一个region 会在纵向上，进行拆分形成多个 store
每个 store保存 就是一张表中的，一个region中的，一个列簇的所有数据
一个store：
    memstore
    blockcache
    storeFile  HFile
```

4.5. HBase的核心概念

架构概念：

```
HMaster          主节点，依赖于zookeper实现HA
HRegionserver    管理region
client           负责发起请求的：能缓存之前已经查询得到的regioin的位置
zookeeper        帮助hbase实现一些服务协调
HDFS             底层文件系统支撑，hbase的数据文件，都存放在 HDFS中，
```

表的物理概念：

```
rowkey           行健，一个行健对应的所有数据，就是一条记录，该记录中可以包含多个（qualifier-value）对
column family    列簇，多个qualifier可以被组织成一个列簇进行集中管理，同时在物理上他们的存储也是分开的
qualifier        列
timestamp        用来表示列的值的版本，越大表示该数据越新
value            值
```

表的逻辑概念：

```
table       表
region      表中个一段rowkey范围中的数据， 都是按照rowkey进行数据排序的
store       如果这张表有多个列簇，那么就会生成多个store
memstore    内存数据
storefile   磁盘文件
blockcache  读缓存
```

核心动作概念：

```
put         插入数据
get/scan    查询/扫描
flush       memstore的数据刷出来，形成磁盘文件
split       当一个region变大的时候（一定的标准：10G），就会一分为二。 分出来的两个regioin
也都是有序
compact     当storefile达到一个个数的时候，就会触发 compact操作：多个HFile合并成一个
```

当 MemStore 达到阈值，将 Memstore 中的数据 Flush 进 Storefile；

compact 机制则是把 flush 出来的小文件合并成大的 Storefile 文件。

split 则是当 Region 达到阈值，会把过大的 Region 一分为二。

将来看源码的时候：

```
1、集群启动
2、读写流程
3、flush, compact, split
```

4.6. HBase的表逻辑模型

维度的概念：

```
f(x) = y                根据key找value          map
f(x1, x2) = y           根据ID和列找value         mysql excel 二维表结构
f(x1, x2, x3) = y       根据长，宽，高，定位立体空间中的某个点      三维结构
.....

HBase表：是一个四维结构
HBsae： NoSQL KeyValue类型数据：  key = qualifier,  value = val
val = f(rowkey, column Family, qualifier, timestamp)
timestamp: 表示 val 的版本的
column Family： 以为一个rowkey中包含的qualifier有可能特别多，这个列簇，根据某个标准，把一
部分的qualifier 放在一起，形成为列的家族
rowkey: 一个rowkey就是一条数据

完整的解释：
一条数据有一个唯一的rowkey，这个rowkey中管理了很多的keyvalue但是这些keyvalue可以被组织成很
多列簇，每个key的value也可以保存很多个版本的值

一张表有很多条数据，有很多个rowkey
一个rowkey有很多个column Family
一个column Family有很多个key
```

一个key有很多value
最终确定value通过timestamp来确定

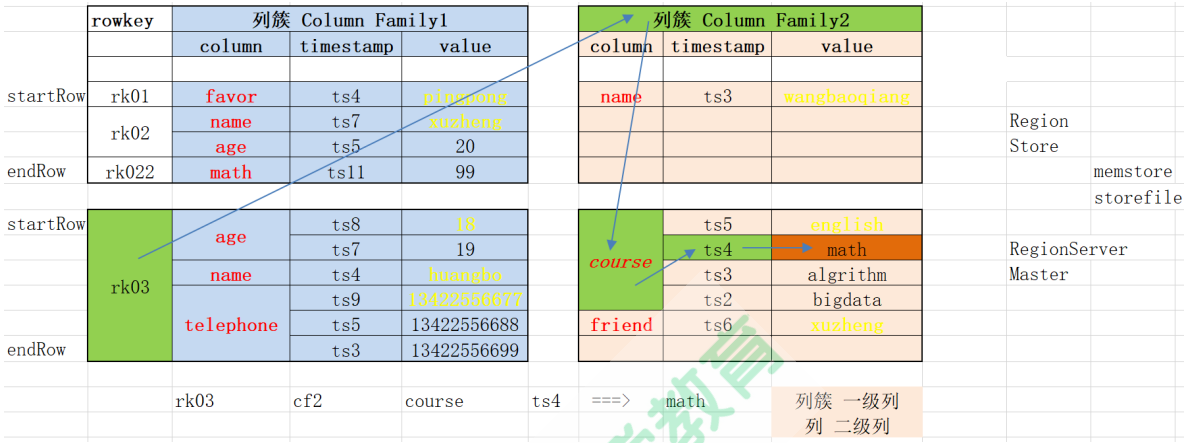
```
table.get(rowkey, cf, qualifier, tiemstamp) = value
```

```
select name from student where id = 1;
用函数表示：
result = f(id=1, qualifier=name)
```

MySQL + Excel 二维表

HBase 四维表

Kylin 多维分析引擎 OLAP cube ND



也可以吧hbase表理解成一个非常复杂的map：

```
table.get(timestamp) = value
table.get(qualifier) = map(timestamp, value)
table.get(cf) = map(qualifier, map(timestamp, value))
table.get(rowkey) = map(cf, map(qualifier, map(timestamp, value)))
table = map(rowkey, map(cf, map(qualifier, map(timestamp, value))))
```

4.7. HBase Java API 设计详解

```
1、配置链接相关：
HBaseConfiguration  Connection  ConnectionFactory

2、管理相关：
Admin  HTable

3、表抽象相关
HTableDescriptor   ColumnFamilyDescriptor

4、增删改查相关：
Put Delete Get Scan

5、数据抽象相关：
ResultScanner  Result   Cell/KeyValue

6、其他：
```

```
Filter  Coprocessor
```

4.8. HBase的核心工作机制读写流程解析

4.8.1. HBase寻址机制

不管你做插入，还是查询，你都需要先确定， 你操作的数据的rowkey 到底属于哪个用户表的那个region，这个region 到底在那个regionserver里面！

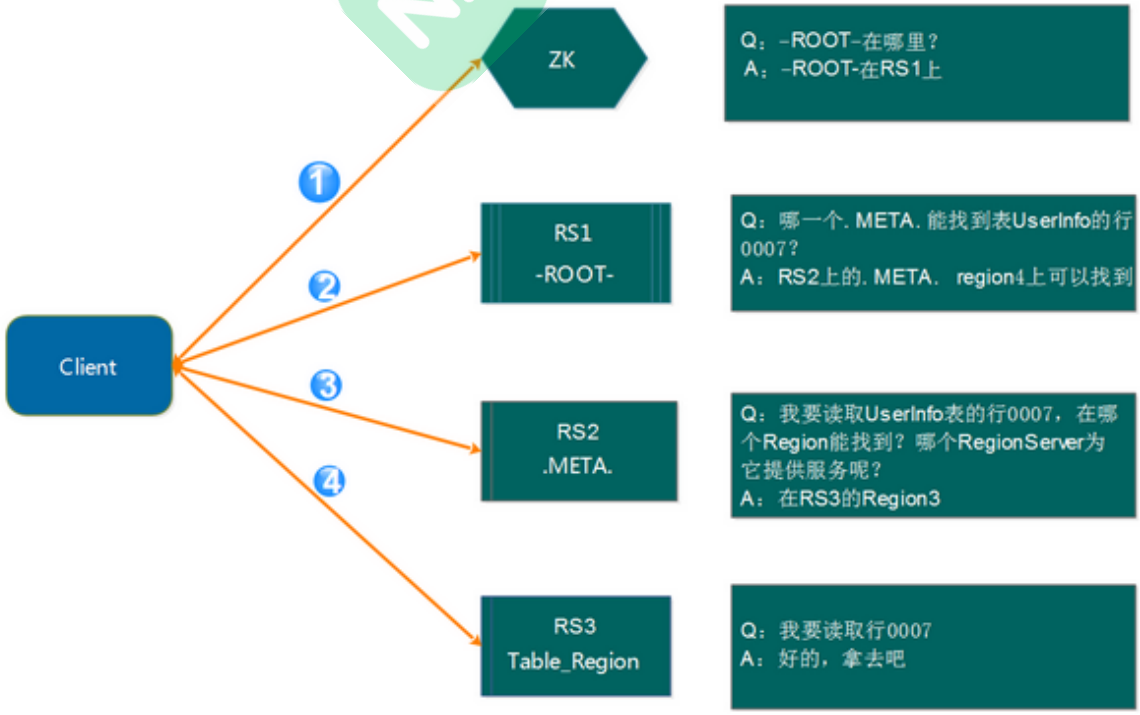
寻址的目的：找到你要操作的数据的workey到底在那个regionserver的region中！

```
老版本：三层结构    zk   root   meta   user
新版本：两层结构    zk          meta    user
```

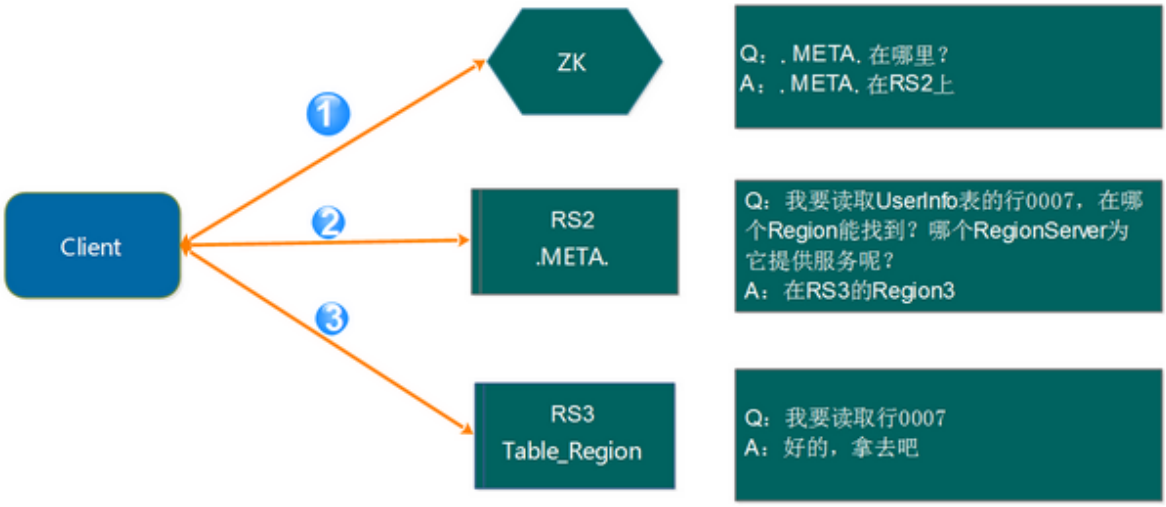
```
要点：
1、root表，meta表，user表的 region 都被regioinserver 管理
2、root 表只有一个region， 这个region的位置，就被存储在zk中
```

```
任何一条数据，一定只有一个确定的region来操作！
这个region（假设是region101）到底在哪里？
1、先访问zk获得 meta表的位置 ， rs1
2、发请求给 rs1，扫描 meta表 的 region ，或者 用户表的region 位置    rs2，同时也确定了是哪个region：假设是 region101  ===> 访问 meta表的region 获得 用户表的region101 到底在哪个regionsever
3、发请求给 rs2，请求的处理，由 region101 来进行
```

老版本寻址机制：



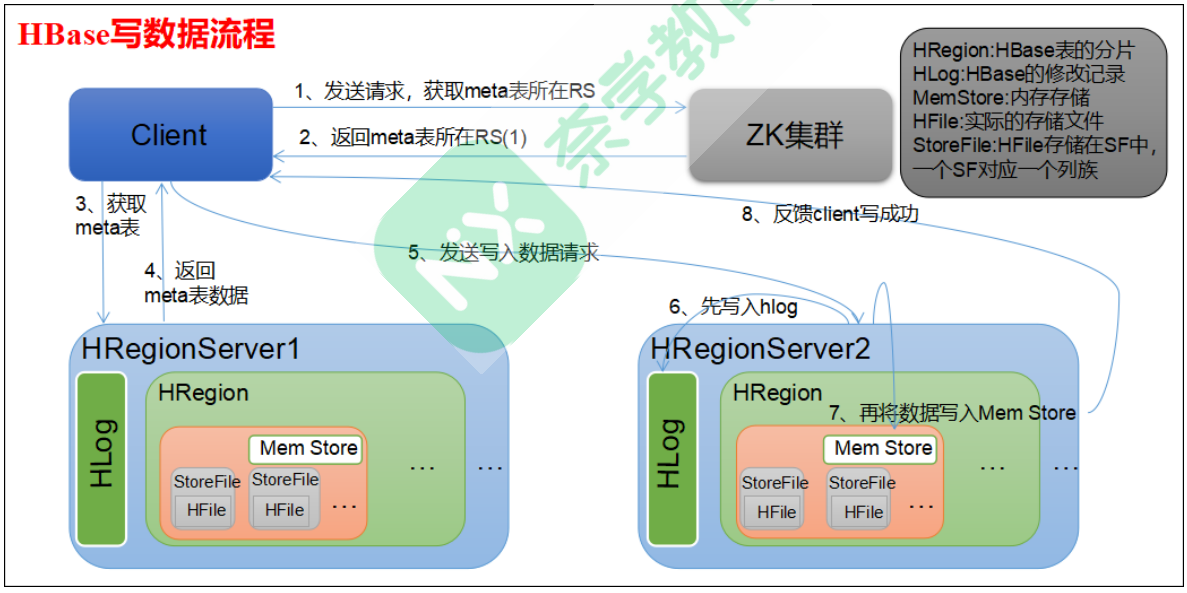
新版本寻址机制：



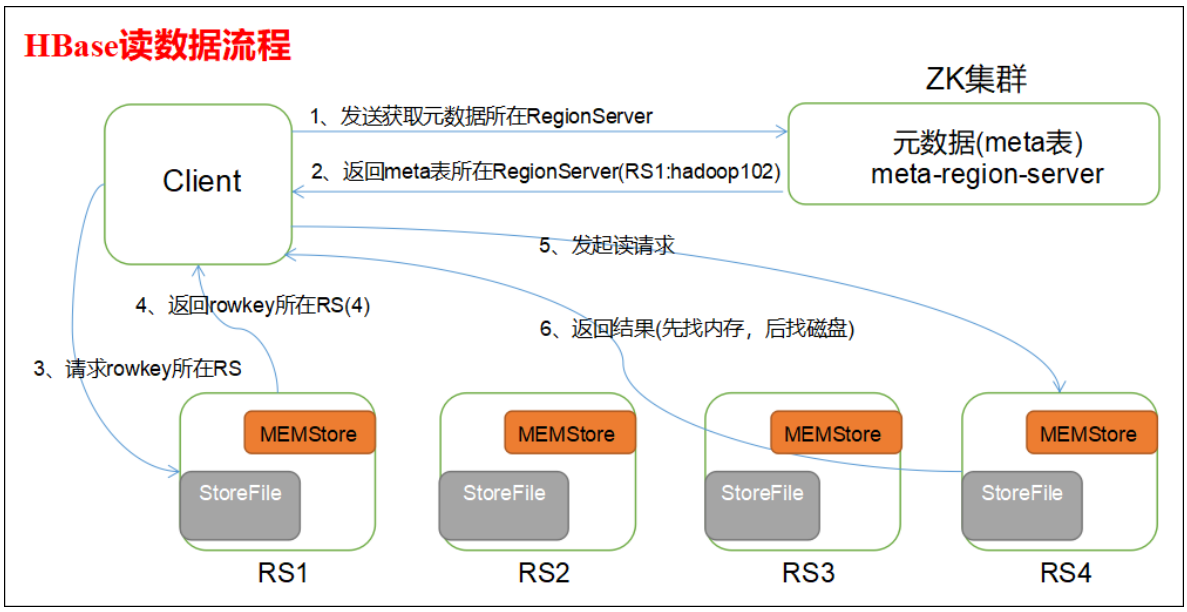
scan 是范围扫描：

```
指定 startRow，  指定 endRow
get 操作返回的是一个 region 的信息
scan 操作返回的是多个用户表的region的位置
```

4.8.2. HBase 写数据流程



4.8.3. HBase 读数据流程



4.9. HBase 的 Flush,Split 和 Compact

当 MemStore 达到阈值，将 Memstore 中的数据 Flush 进 Storefile；

compact 机制则是把 flush 出来的小文件合并成大的 Storefile 文件。

split 则是当 Region 达到阈值，会把过大的 Region 一分为二。

5. 总结

重点是告诉你：如何设计一个HBase中实现低延迟的随机读写的数据库系统